Flaw Repairing in Images Using Contour Representations

http://www.wam.umd.edu/~zryuan/390/

Amy Yuan

Advisor : Dr. Dave Mount

Abstract

Image processing has long been an important aspect of the computer science world and flaw repairing is one of its crucial components. Flaw repairing is the removal of part of an image and the restoration of a smoothly interpolated subimage. This may be used to remove a blemish on your face or to airbrush yourself out of the picture on the desk of once-your-foe-now-your-boss, like George Costanza. The question is how to accomplish it efficiently. The conventional way has been to manually fill the pixels in the flaw with color levels in the vicinity. A more efficient solution has been proposed by Tetsuo Asano. His method exploits algorithmic techniques from computational geometry, which uses the alternative contour representation of a graph.

Traditionally graphs have been represented as digital matrices, with each entry representing a pixel value. Throughout this paper, color images will be treated as three monochromatic images, depending on the intensity of the red, green and blue components. The intensity of a pixel is indicated by an integer i, in the range 0 to 255. Since the image is treated as being monochrome, these are called gray levels. An alternative representation is based on the use of contour lines. For each gray level i we compute connected regions of pixels with gray levels greater than or equal to i. Representation of an image as a collection of those boundary lines associated with gray levels (contour lines) is the contour representation of an image [1].

We first give an efficient algorithm for computing contour lines for each gray level. Then we discuss how to accomplish flaw repairing using this information.

Introduction

Let G = g[i][j] i = 1…h-2, j = 1…w-2 be an image such that each g[i][j] is an integer between 0 and L-1. For convenience we embed the image into a two-dimensional h x w array so that the boundary elements are all –1. A one-dimensional array p[k], k = 0…wh-1 representing the pixels is mapped to the matrix, k = i*w + j.

The horizontal edge below a pixel p[k] and vertical edge to the right of p[k] are used to represent the pixel and numbered as 2k and 2k+1, respectively. All edges are sequentially numbered, e_i, i = 0…2wh –1. Each edge is directed so that the interior of a region lies to its right. North and East are labeled positive while South and West as negative.

The set of connected pixels with gray levels greater than or equal to i, $0 \le i \le L-1$, makes up the region R_i. By the definition, $R_{i+1} \subseteq R_i$ for every i. Each connected region may have multiple boundary components because of holes.

Contour[i], $0 \le i \le L-1$, is the set of edges whose corresponding pixels have gray levels greater than or equal to i. The flaw region is a rectangular area within the image.

We have implemented a program performing flaw repair in C++ and made experiments on four different images, which will be described later. Inputs to the program include a raw ppm format image file and the coordinates of the four corners of the flaw region to be restored. Three matrices representing the red, green and blue level values of the pixels are constructed from the image file and used as the primitive image information throughout the program. The output consists of the new image in raw ppm format.

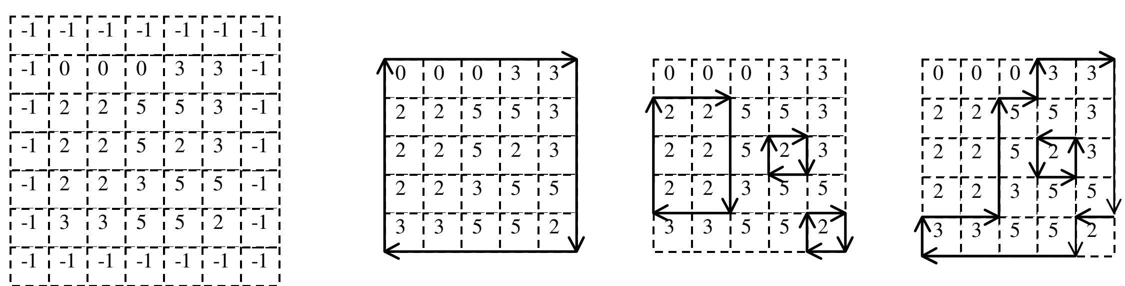
First, we will present an algorithm to construct the contours. Then contour lines inside the flaw region are removed and thus result in the disconnection of some contours. The question is how to connect the disconnected contours, ie, the edges intersecting the boundary of the flaw region, in a natural way. We apply dynamic programming to find the minimum total length to interconnect these edges and then fill the flaw with new contour lines. Finally, a new image is constructed from these contour lines, and output as the final result.

Algorithms

Construct the Contours

Lemma: Let e_i be an edge between two adjacent pixels p and q with gray levels g[p] and g[q], respectively. Then, the edge is included in the boundaries of the regions $R_{\min(g[p],\, g[q])+1} \ldots R_{\max(g[p],\, g[q])}$ if $g[p] \neq g[q]$.

As the first step we perform a raster scan to detect edges. For an edge e_i between gray levels g_1 and g_2 with $g_1 < g_2$, we add edge e_i to the list contour[i] where the direction of the edge is determined so that the pixel of greater gray level lies to the right of the directed edge. After the raster scan over the whole image, the list contour[0] consists of all edges on the boundary of the region R_0, which in this case all together forms a rectangle enclosing the original image.



Original Matrix Contour for Level 0 Contour for Level 2 Contour for Level 3

Figure 1: Contours.

Calculate Intersections

For each two adjacent pixels p and q along the boundary of the flaw, if $g[p] \neq g[q]$, then edges with level min (g[p], g[q])+1 … max(g[p], g[q]) must be included in the list of intersecting edges. The number of incoming edges must be equal to the number of outgoing and hence the total number of intersections must be even.

DP Formulation

An algorithm for flaw repairing is as follows: Let $(v_1, v_2 \ldots v_n)$ be a sequence of terminals on the boundary of the flaw. If v_i and v_j have different directions (incoming/outgoing) and the same gray level, d(i, j) is defined to be the Euclidean distance between v_i and v_j. Otherwise, d(i, j) is defined to be $+\infty$. τ (i, j) is defined to be the minimum among sums of lengths of legal paths interconnecting $v_i, v_{i+1} \ldots v_j$ which satisfy the conditions stated above. If there is no legal interconnecting pattern then τ (i, j) is $+\infty$. The goal is to find τ (1, n).

$$\tau(i, j) = 0 \qquad \text{if } j \le i.$$
$$\tau(i, j) = \min_{i+1 \le k \le j} \left(d(i, k) + \tau(i+1, k-1) + \tau(k+1, j) \right) \qquad \text{if } j > i \text{ and both } k-i \text{ and } j-i \text{ are even.}$$

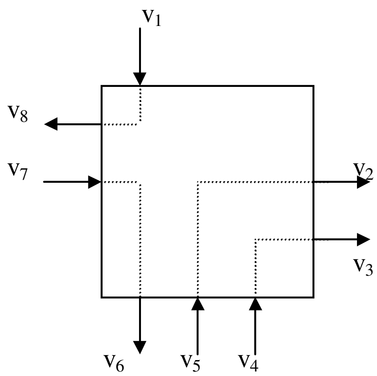


Figure 2: DP formulation.

Bresenham's Algorithm

To connect two edges, we apply Bresenham's line algorithm.

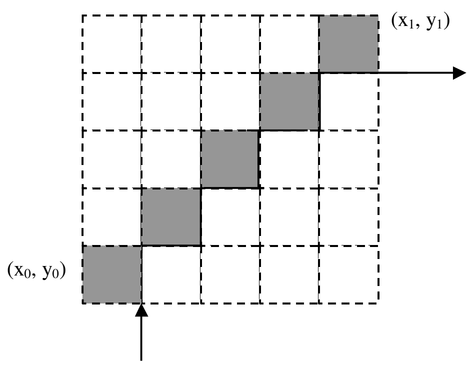


Figure 3: Bresenham's algorithm.

```
void BresenhamLine (                    /* Assume -1≤ m ≤ 1, x0 < x1 */
        int x0, int y0,                 /* Left endpoint */
        int x1, int y1,                 /* Right endpoint */
        int value)                      /* Value to place in line's pixels */
{
        int x;                          /* x runs from x0 to x1 in unit increments. */

        double dy = y1 - y0;
        double dx = x1 - x0;
        double m = dy / dx;
        double y = y0;

        for (x=x0; x≤x1; x++)
        {
                WritePixel (x, Round(y), Value);    /* Set pixel to value */
                y += m;                             /* Step y by slope m */
        }
}
```

We assume that the line's slope is between –1 and 1. Other slopes can be handled by suitable reflections about the principal axes [3].

This is a standard line drawing algorithm used in computer graphics. We have modified the algorithm to draw a line along the cracks between pixels, rather than through the pixels themselves.

Back to Matrix

How can we use the new contour lines inside the flaw to determine the pixel values? A pixel p[k] may be enclosed in several regions and its adjacent edges included in several contours of different levels. To decide what value the pixel should have, we examine the edges it is associated with. If p[k] has positive edges $e_i \ldots e_j$ to its right or beneath it, p[k] = min (levels of $e_i \ldots e_j$) –1. If p[k] has negative edges $e_i \ldots e_j$, p[k] = max (levels of $e_i \ldots e_j$).

However, a pixel may not have any adjacent edges that are on the boundary of some contour. Within the flaw, we then propagate the pixel values as determined from above to its undetermined neighbors.

Lemma: Initialize all p[] within the flaw to be $+\infty$. Let p[k] be a pixel with unknown value in the flaw region. If p[k] has right and left neighbors p[i] and p[j], respectively, (not necessarily adjacent) and both values have been determined, p[k] = min (p[i], p[j]). If no such p[i] exists, let p[m] be the pixel just outside the left boundary and on the same row as p[k], p[k] = p[m]. If no such p[j] exists, let p[n] be the pixel just outside the right boundary and on the same row as p[k], p[k] = p[n].

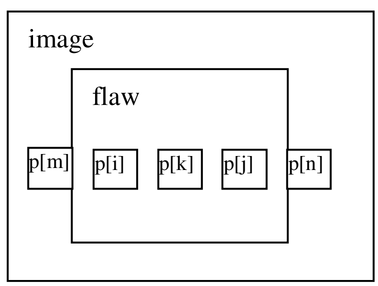


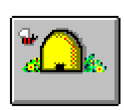
Figure 4: Propagate pixel values.

Results

The total running time is bounded by the dynamic programming calculation, or $\theta(n^3)$, with n being the total number of intersections.

Flaw repairing is trivial if the vicinity of the flaw region has no variation in colors. The total number of intersections will be zero and all pixels within the flaw will be given the value of pixels outside the boundary. The result appears seamless.

Original image



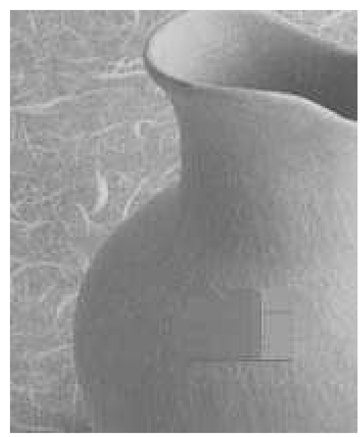
New image after removing "Home"

Figure 5: Plain background.

However, if the flaw region lies on top of a textured surface, with subtle variations in colors, the result is not always optimal. After computing the shortest total cost of interconnecting all the incoming/outgoing edges, the original texture may not be preserved in the new image (see Figure 7 and 8).

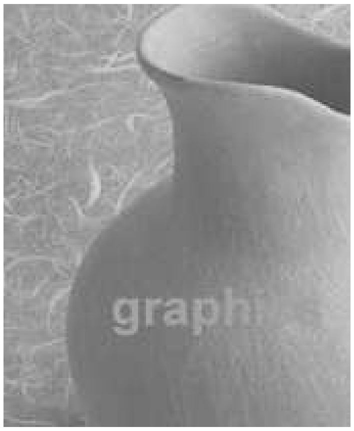
Original image New image after removing the leaf in the bottom right corner

Figure 6: Texture background.

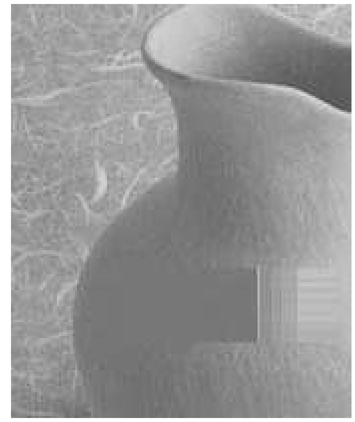


Original image New image after removing "UM"

Figure 7: Complex background with small flaw.



Original image



New image after removing "graphics"

Figure 8: Complex background with big flaw.

Conclusions and Future Study

Limitations of the program include the restrictions on the input format and not working satisfactorily on all types of images as shown above. Further work can be done to accept arbitrarily shaped flaw region and to enhance the results by finding a better total cost function in the dynamic programming calculation. However, all flaw regions can be subdivided into rectangles and the rules of this program still apply.

References

[1] T. Asano and S. Kimura, "Contour Representation of an Image with Applications" in *Vision Geometry IV*, R. A. Melter, A.Y. Wu, F. L. Bookstein, W.D.K. Green, Editors, Proceedings of SPIE 2573, 14-22 (1995).

[2] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1990.

[3] J. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics: Principles and Practice (Second Edition in C)*, Addison-Wesley, Reading, MA, 1996.

[4] D. Hearn, M. P. Baker, and P. M. Baker, *Computer Graphics*, Prentice Halls, Englewood Cliffs, NJ, 1994.

[5] D. M. Mount, *CMSC451 Lectures Notes*, http://www.cs.umd.edu/~mount/451/, Fall 1998.

Appendix : Code for Dynamic Programming Algorithm

```
/***************************************************************************

Function Purpose: To compute the min total cost to connect all intersections.

The list intersects[] stores the intersecting edges. The list is constructed
      by walking around the boundary of the flaw clockwise, adding edge(s)
      whenever two adjacent pixels have different pixel values. Then, do a
      stable sort on the list by the pixels' gray levels.
The integer count is the total number of intersections.
The 2-d array d[i][j] stores either the Euclidean distance between
      intersects[i] and intersects[j] when there is a legal connection
      between i and j or INF otherwise.
The 2-d array t[i][j] stores the min total cost to interconnect edges from
      i through j.
The 2-d array m[i][j] stores the index of the first cut when connecting
      from i through j.
The 1-d array mid[i] stores the index of the edge in the list intersects[]
      that i will be connected to.

The loop below following this function will connect all pairs of
      incoming/outgoing edges as computed:
      for (int i=0; i<count; i++)
            connect (intersects[i], intersects[mid[i]]);

***************************************************************************/

void Graph::CalShotestDist ()
{
      int i, j, m, k;
      int **d = new int *[count];
      int **t = new int *[count];
      int **m = new int *[count];
      mid = new int [count];

      /* initialization and space allocation */

      for (i=0; i<count; i++)
      {
            mid[i] = -1;
            d[i] = new int [count];
            t[i] = new int [count];
            m[i] = new int [count];
      }

      for (i=0; i<count; i++)
      {
            for (j=0; j<count; j++)
            {
                  m[i][j]=-1;

                  if (intersects[i].dir == intersects[j].dir ||
                        intersects[i].level != intersects[j].level)
                        d[i][j] = INF;
```

```
                else
                      d[i][j] = abs(intersects[i].x - intersects[j].x)
                            + abs(intersects[i].y - intersects[j].y) ;
          }
    }

    for (i=0; i<count; i++)
    {
          for (j=0; j<count; j++)
          {
                if (i>=j)   t[i][j] = 0;
                else        t[i][j] = INF;
          }
    }

    /* DP calculation  */

    int temp;
    for (m=2; m<=count; m+=2)
    {
          for (i=0; i<=count-m; i++)
          {
                j = i + m - 1;

                for (k=i+1; k<=j; k++)
                {
                      temp = d[i][k] + t[i+1][k-1] + t[k+1][j];
                      if (temp<t[i][j])
                      {
                            t[i][j] = temp;
                            m[i][j] = k;
                            m[j][i] = k;
                      }
                }
          }
    }

    FindMid (m, i, j);

    /* space deallocation */

    for (i=0; i<count; i++)
    {
          delete [] d[i];
          delete [] t[i];
          delete [] m[i];
    }

    delete [] d;
    delete [] t;
    delete [] m;
}
```

```
/*****************************************************************************

      Function Purpose: To construct mid[] from m[][].

*****************************************************************************/

void Graph::FindMid (int **m, int i, int j)
{
      if (i>=j)         return;

      /* m[i][j] is the index of the first cut when connecting i through j */
      int k = m[i][j];

      if (k==j)   /* the first connection is connecting i w/ j */
      {
            mid[i] = j;
            mid[j] = i;
            FindMid (m, i+1, j-1);
      }
      else        /* the first cut is between i and j */
      {
            mid[i] = k;
            mid[k] = i;
            FindMid (m, i+1, k-1);
            FindMid (m, k+1, j);
      }
}
```